5921 44

On the Information Content of Program Traces

Michael Frumkin, Robert Hood and Jerry Yan[1]

NAS Technical Report NAS-98-008 March 98

{frumkin,rhood,yan}@nas.nasa.gov
NAS Parallel Tools Group
NASA Ames Research Center
Mail Stop 258-6 or T27A-1
Moffett Field, CA 94035-1000

Abstract

Program traces are used for analysis of program performance, memory utilization, and communications as well as for program debugging. The trace contains records of execution events generated by monitoring units inserted into the program. The trace size limits the resolution of execution events and restricts the user's ability to analyze the program execution. We present a study of the information content of program traces and develop a coding scheme which reduces the trace size to the limit given by the trace entropy. We apply the coding to the traces of AIMS instrumented programs executed on the IBM SP2 and the SGI Power Challenge and compare it with other coding methods. Our technique shows size of the trace can be reduced by more than a factor of 5.

[1] MRJ Technology Solutions, Inc., NASA Contract NAS2-14303, Moffett Field, CA 94035-1000

1. Introduction

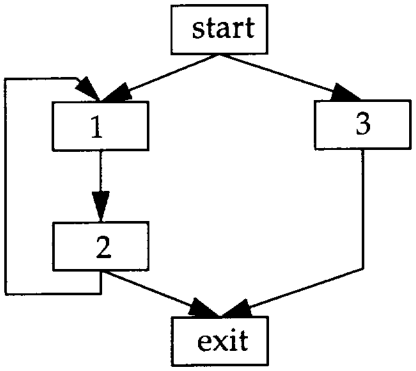
A program trace contains records of the events that happened during the program execution. Each record contains the event identifier, location and the time when it happened. Depending on the event type, the record may contain additional information such as a message tag and a message destination. The trace records are generated by the monitoring units inserted into the program. This insertion can be done with an instrumentation tool into the source code [4], into assembly code or even into a loaded and running program [11]. An analysis of trace records gives data on the program performance, memory utilization and helps in the program debugging [13]. There are several tools for program instrumentation, monitoring and trace visualization: `prof`, `gprof`, `pixie` [3], AIMS [4], Paradyn [11].

The larger the number of processors and the finer the event resolution, the larger the program trace. The trace size (common trace size is a dozen megabytes) limits the user's ability to monitor execution events and to localize the statements causing problems with the program. A number of studies were done to reduce the trace size [6]. In this paper we use an information-theoretic technique to reduce average trace size to the theoretical low bound given by the trace entropy. Theoretically, the technique is based on *the Noiseless Coding Theorem* [1] asserting that an average code length of a random variable cannot be less than the entropy of this variable. The appropriate code length can be achieved by using the Huffman coding or by using the dynamic Huffman coding [9,10] if the distribution is unknown *a priory*. In practice, this technique is based on the collection of an event histogram and on the application of the Huffman coding. For the set of records considered in this paper, the trace entropy is the sum of the entropy of the program Markov chain, the communication entropy and the entropy of time stamps. We give a method for calculation of the three components of trace entropy and compare the entropy with results of applying the standard compression technique (`compress` and `gzip`). For the four traces we considered our compression is better than `gzip` and reduces the size of trace by a factor of 5.

2. The Program Graph

Depending on the abstraction level, a sequential program can be represented by its *call graph* [3] or *flow graph* [2]. In this paper we use the flow graph representation. The vertices of the flow graph are basic blocks of the program code, and arcs are the possible ways for the program counter (pc) to move between basic blocks. If the pc is pointing to a basic block *i*, we will say that program is in the state *i*. A transition of the program from one state to another will be referred as an *event* by saying that event *i->j* occurred if the pc moves from state *i* to state *j*.

An execution of a sequential program can be represented by a path in the flow graph. The path starts at the first statement of the main function (*start*) and terminates at the *exit*. If a program is in a state *i* and event *i->j* occurrs we add the arc *(i,j)* to the path. The length of the execution is the number of arcs in the path, see Figure 1. If only a subset of states is monitored and only transitions between this reduced set of states are recorded, the flow graph can be reduced to a smaller graph by discarding nonmonitored states and by adding arcs reflecting possible transitions between the reduced set of states. This reduced flow graph will be referred as the *program graph*. We will refer to the nodes of the graph as states.



Path = start,1,2,1,2,1,2,exit

FIGURE 1. A program graph and an execution path

We will consider traces of message passing parallel programs. An execution of each process of a parallel program can be represented by a path in the program graph as described above. In a message passing program pairs of processes interact by exchanging messages. This interdependency can be specified by a sequence of pairwise matching send/receives.

The coding of traces of sequential programs is considered in sections 3 and 4. We consider coding of messages of message passing programs in section 6. Coding of time stamps in parallel programs is considered in section 7. Results of calculations of the minimal length of AIMS traces are tabulated in section 8.

3. Number of Possible Traces

A simple lower bound for the length of a lossless code of a trace can be obtained by comparing the number of traces with the number of codes. There are at most $O(2^L)$ codes of length L in the {0,1} alphabet, hence the maximum code length can not be less than the logarithm of the number of traces with N records[1].

The number of traces with N records is the number of directed paths in the program graph G from *start* to *exit* of length N. This number can be estimated through eigenvalues of the adjacency matrix of G:

$$A = [a_{ij}],\ 1 \le i, j \le n$$

where n is the number of states, $a_{ij} = 1$ if there is an arc from the state i to the state j and $a_{ij} = 0$ otherwise. The number of paths from i to j of length N equals the ij^{th} element of A^N, see [7]. Let

$$A = U^{-1} \Lambda U$$

be a spectral decomposition of A, where Λ is a diagonal matrix with λ_1, $\lambda_2, \ldots, \lambda_n$, $|\lambda_1| \ge |\lambda_2| \ge \ldots \ge |\lambda_n|$ elements on its main diagonal[2]. Then $A^N = U^{-1}\Lambda^N U$ and for the number of paths of length N from i to j we have a formula:

$$a_{ij}^N = \lambda_1^N v_{i1} u_{1j} + \ldots + \lambda_n^N v_{in} u_{nj}$$

where v_{ik} and u_{kj} are elements of matrices U^{-1} and U respectively. Hence, $\log a^N_{ij} = O(N \log |\lambda_1|)$ and $\log a^N_{ij} = \theta(N \log |\lambda_1|)$ if we assume that v_{i1} and u_{1j} are nonzero[3].

The maximal module λ of eigenvalues of the adjacency matrix can be estimated by means of maximal and minimal in- and out- degrees of the states:

$$Max\left\{Min\left\{d_i^{in}\right\}, Min\left\{d_i^{out}\right\}\right\} \le \lambda \le Min\left\{Max\left\{d_i^{in}\right\}, Max\left\{d_i^{out}\right\}\right\}$$

[1] All *logs* are on base 2 in this paper

[2] If A has nontrivial Jordan blocks our arguments can be modified appropriately without affecting the result.

[3] $f(n) = \theta(g(n))$ if there are constants c and C such that $cg(n) < f(n) < Cg(n)$. $\theta(g(n))$ is a class of functions and we use notation $f(n) = \theta(g(n))$ to indicate that f is in this class

In a typical program graph, degrees of majority of states are bounded by a constant; and only few states have larger degrees. This property can exploited for tighter bounding of the maximum module of the eigenvalues. If, for example, one state has (large) degree D, and degrees of all other states are bounded by b, then using Gershgorin Circle Theorem, [12] we can get a sharper upper bound for the largest eigenvalue:

$$\lambda \le \frac{b}{2} + \sqrt{D + \left(\frac{b}{2}\right)^2}$$

We can conclude that the minimum code length of a trace in the worst case is the logarithm of the number of possible N event traces and can be estimated as

$$L(N) = \theta(N \log \lambda)$$

where λ is the maximal modules of eigenvalues of the adjacency matrix of the program graph.

4. The Entropy of Traces

The transition of the pc from one state to an another can be considered as a stochastic event. It means that each trace T has a probability $p(T)$ to appear as a trace of a program. We want to minimize the expected length of trace code:

$$\sum_T c(T)p(T) \rightarrow Min$$

where $c(T)$ is the code length of T.

The minimum can be obtained from the Noiseless Coding Theorem [1, p. 37] which states that for any lossless coding of a random variable with distribution $\{p_i\}$ the minimum average code length can not be less than the entropy of this distribution, that is, $H = -\Sigma p_i \log p_i$. This bound can be closely approached with a code having the i^{th} code word length of $-[\log p_i] + 1$ see [1, p. 39].

It follows that the minimum of expected length of trace code is close to the entropy of the set of traces $\{T\}$. We will express the probability $p(T)$ through the transition probabilities of the program. For the probability p_j^{n+1} of a path of length $n+1$ to terminate in a state j we have a relation:

$$p_j^{n+1} = \sum_i p_i^n p_{ij}$$

where p_{ij} is the transition probability from state i to state j. The relation holds in the case when p_{ij} is independent of the path by which the state i was reached as well as on the value of n. This property is known as *Markov property* [1]. For now we will assume that the program has this property. At the end of the section we will discuss a modification of state definitions and enhancements to the program model which can be done in the case where the state transitions lack the Markov property.

In the matrix form, the relation above can be written as follows: $(p^{n+1})^t = (p^n)^t Q$, where p^n is the probability column vector, $Q = [p_{ij}]$ is the matrix of transition probabilities and the superscript t means the transposition. If we consider a Markov chain [1] with matrix Q as a transition matrix then p^n converges to a steady state probability vector of the Markov chain. The steady state vector w is defined as the left eigenvector of Q with eigenvalue 1:

$$w = wQ$$

If Q is an irreducible matrix the Perron-Frobenius theorem [8, p. 399] asserts that w exists and is unique.

In addition to the stationary distribution, we need the entropy of each state i:

$$H(i) = -\sum_j p_{ij} \log p_{ij}$$

If x is a sequence of states then, T_x will denote a trace with suffix x. The superscript of trace T^n denotes the number of events in the trace.

Now we can write a recurrence relation for entropy of traces of length $n+1$ through the entropy of traces of length n:

$$H_{n+1} = -\sum_{T^{n+1}} p(T^{n+1}) \log p(T^{n+1}) = -\sum_j \sum_{T_j^{n+1}} p(T_j^{n+1}) \log p(T_j^{n+1}) =$$

$$-\sum_j \sum_i \sum_{T_{ij}^{n+1}} p(T_{ij}^{n+1}) \log p(T_{ij}^{n+1}) = -\sum_j \sum_i \sum_{T_i^n} p(T_i^n) p_{ij} (\log p(T_i^n) + \log p_{ij}) =$$

$$-\sum_i \sum_{T_i^n} \sum_j p(T_i^n) p_{ij} (\log p(T_i^n) + \log p_{ij}) =$$

$$-\sum_i \sum_{T_i^n} \left(p(T_i^n) \log p(T_i^n) \sum_j p_{ij} + p(T_i^n) \sum_j p_{ij} \log p_{ij} \right) =$$

$$-\sum_T p(T^n) \log p(T^n) - \sum_i \left(\sum_{T_i^n} p(T_i^n) \right) H(i) \approx H_n + \sum_i w_i H(i)$$

Here we use the above mentioned Markov property of the chain:

$$p(T_{ij}^{n+1}) = p(T_i^n)p_{ij}$$

and the property of the steady state vector of Markov chain: for large n

$$w_i \approx \sum_{T_i^n} p(T_i^n)$$

meaning that the probability of state i is equal to the probability of arriving to i in n steps.

From this relation it follows that for large n we can express the entropy of traces trough a steady state vector of a Markov chain and the entropy of its states:

$$H_n \approx n \sum_i w_i H(i)$$

which gives us a lower bound for the expected length of the trace code.

If the Markov property is not true, then we can consider a more general program model. It will lead us to a similar formula for the trace entropy; however, to compute the entropy on right hand side is more difficult. The states of this more general model will be sequences of l program states $u = i_1 i_2 ... i_l$ and events will be transitions $i_1 i_2 ... i_l$ -> $i_2 ... i_l i_{l+1}$. If the value of l is such that the program does not remember how it got in state u then the Markov property will be true for the model and arguments similar to one above can be applied.

5. The Ergodicity of a Program

How accurately does a program trace represent other possible executions of the program? If a set of program inputs can be classified into different categories and for different categories the program behavior varies, then a trace on one input tells you a little about traces on inputs from a different category. In order to get correct transition probabilities, a mixture of traces of runs on different category inputs is necessary.

A program is called *ergodic* if the transition probabilities are independent of program execution. Coding of traces of an ergodic program can be significantly simplified. The transition probabilities can be estimated from the trace of preliminary program execution. These probabilities can be used for generating the Huffman code of the trace records of other executions of the program.

6. The Communication Entropy of Parallel Message Passing Programs

We confine our considerations to single-threaded processes communicating by passing messages. Each process can be described by a program graph as explained in section 2. Transitions between the states in different processes are interdependent through exchanging messages. This causality relation between events in different processes can be uniquely reconstructed if the program trace contains a set of pairwise matchable sends/receives.

We will assume that the message passing program is MPI compliant, meaning that the messages can be matched using MPI *progress* and *order* rules, [4, pp.30-31].

- *Progress:* "If a pair of matching send and receives have been initialized on two processes, then at least one will complete..."
- *Order:* If two messages with the same tag are sent from the same source to the same destination, then they are received in the same order they were sent (the symmetrical property is true for the receiver).

The first rule implies that if there are sends matching a receive, then the receive must be matched with one of the sends (symmetrical for sends). From the second rule it follows that if there are several sends matching a receive, then the receive will be matched with the first posted send (symmetrical for sends). The second rule also implies that the message passing program is deterministic if wild card MPI_ANY_SOURCE and calls MPI_CANCEL and MPI_WAITANY are not used and no MPI error occurred.

These rules together give rise to a unique way to match messages using the process id, message tag and message order. Let S_{ij} and R_{ji} be lists of sends and receives with source *i* and destination *j*. The matching can be done by the iteration of the following step:

- Take the first receive *r* in R_{ji} and find first matching send *s* in S_{ij} (since the messages already have the same source and the same destination it means we take the first send with the same tag). Remove *r* and *s* from the lists.

Tagging messages allows changing the order in which the messages are received. Two messages received in an order different from which they were sent are called intertwined, cf. [4, p 31]. An appropriate tagging allows receiving messages in an arbitrary order. In practice, intertwining is limited by the size of system buffers. An example of an intertwined sequence of messages is shown in Figure 2. For the trace records of messages it means that the record should contain the message destination/source and the message tag. Otherwise the causality relation of events in different processes will not be uniquely reconstructible from the trace records.

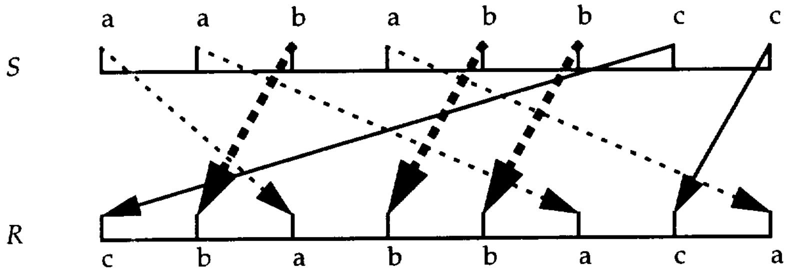


FIGURE 2. Intertwining of two sequences by using tags *a,b* and *c*

We assume that the message passing program is deterministic. The program sends/receives can be matched uniquely if for each pair of processes *i* and *j*, a chronological list S_{ij} of tags of messages sent by *i* to *j* and a chronological list R_{ij} of tags of messages received by *i* from *j* are specified. These lists can be composed into a send matrix $S = [S_{ij}]$ and a receive matrix $R = [R_{ij}]$ of the program. These matrices specify uniquely the causality relation between events of the program.

A pair of send/receive matrices is *consistent* if the length s_{ij} of the list S_{ij} is equal to $r_{ji,}$ the length of the list R_{ji}, and the number t_{ijk} of tags with value *k* is the same in both lists. In other words the matrix $| r_{ij} |$ is a transposition of the matrix $| s_{ij} |$ and the list R_{ji} is a permutation of the list S_{ij}. Any consistent pair of matrices can arise as a send/receive matrix pair of the program (it is sufficient to set message tags according to the list elements).

Now we fix $s_{ij} = r_{ji}$ and t_{ijk} and count the number of possible consistent *S* an *R* matrices. As in section 3 the logarithm of this number will give us the minimum code length of the communication matrices of the program.

Let s_i be the total number of sends in process *i*, $s_i = s_{i1} + s_{i2} + ... + s_{ip}$, where *p* is the number of processes. For the given $s_{i1}, s_{i2}, ..., s_{ip}$ the total number of possible send sequences in the process *i* (which is the *number* of possible instances of i^{th} row of the matrix *S*) is

$$S_i = \frac{s_i!}{\prod_j s_{ij}!} \prod_j^p t(s_{ij})$$

where

$$t(s_{ij}) = \frac{s_{ij}!}{\prod_k t_{ijk}!}$$

is the number of possible lists of length s_{ij} having t_{ijk} tags with value *k*. The send sequences in different processes are independent so the total number of send lists equals ΠS_i. A similar formula for the j^{th} row of matrix *R* is true:

$$R_j = \frac{r_j!}{\prod_i s_{ij}!} \prod_i^p t(s_{ij})$$

where $r_j = s_{1i} + s_{2i} + \ldots + s_{pi}$.Let $M = s_1 + s_2 + \ldots + s_p = r_1 + r_2 + \ldots + r_p$ be the total number of messages in the program.

The total number of send/receive matrices equals $\Pi S_i \Pi R_j$. The logarithm of this number can be well approximated with the Stirling formula as:

$$M(H_s + H_r + 2H_{st})$$

where

$$H_s = -\frac{1}{M}\sum_i^p \sum_j^p s_{ij} \log\frac{s_{ij}}{s_i} = -\sum_i^p \frac{s_i}{M} \sum_j^p \frac{s_{ij}}{s_i} \log\frac{s_{ij}}{s_i}$$

$$H_r = -\frac{1}{M}\sum_i^p \sum_j^p s_{ji} \log\frac{s_{ji}}{r_i} = -\sum_i^p \frac{r_i}{M} \sum_j^p \frac{s_{ij}}{r_i} \log\frac{s_{ij}}{r_i}$$

are the entropy of the program sends and receives respectively and

$$H_{st} = -\frac{1}{M}\sum_{i}^{p}\sum_{j}^{p}\sum_{k}^{K} t_{ijk}\log\frac{t_{ijk}}{s_{ij}} = -\sum_{i}^{p}\sum_{j}^{p}\frac{s_{ij}}{M}\sum_{k}^{K}\frac{t_{ijk}}{s_{ij}}\log\frac{t_{ijk}}{s_{ij}}$$

is the entropy of message tags in the program, where K is the number of tags.

We can conclude this section saying that the minimal code length of communication events of a message passing program equals to the number program messages times the sum of the communication entropy of the program and of the entropy of the message tags. The last two numbers can be easily computed if the trace records contain the message source, destination and message tags, cf. Table 3.

7. Time Stamps in Program Trace

Assume that the sequence of trace events is known and we want to specify the time when the events occurred. The direct recording of the time stamps expressed in counts of clock periods or time resolution of the clock counter would give rise to the code of length

$$L = \sum_{i} N_i \log T_i$$

where T_i is the execution time of i^{th} process and N_i is the number of events in the trace of i^{th} process. Should we record the elapsed time between events we would have a similar formula for the code length of the time stamps with T_i replaced by the maximum time between events in i^{th} process which can be substantially smaller than T_i. The use of an entropy efficient coding would reduce the code length to

$$L = \sum_{i} N_i H_i$$

where H_i is the entropy of the distribution of time intervals between events.

The time stamps are used for evaluation of the program performance, ordering of the program events and for displaying of program traces. Neither of these applications requires an exact time and can tolerate approximate values of time stamps as long as the approximation does not change event ordering and event resolution. For the approximation, it is essential not to move events relative to each other significantly. This requirement can be formulated in terms of event chronology defined as a chronologically ordered sequence of times of events in all processes of the program. Let x_{kl} be the time interval between neighbor events k and l and τ_l be the time interval between neighbor events to l in event chronology, see Figure 3

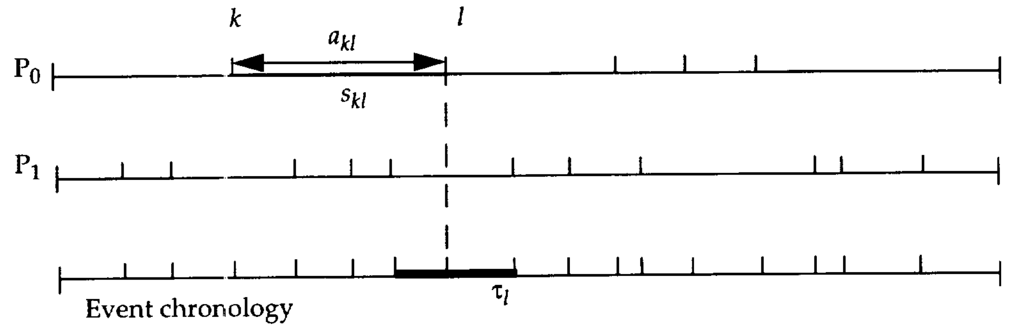


FIGURE 3. Combining events on different processes into event chronology

We will require that the approximation a_{kl} of the time stamp x_{kl} of any event is within ε times the length of the interval between neighbor events:

$$|x_{kl} - a_{kl}| \le \varepsilon\tau_l$$

Let $p_i = P\{s_{kl} = i\}$ and $r = \varepsilon\tau$. If we use the approximation of time stamps with the precision ε, then instead of the random variable x with the distribution $\{p_i\}$ we have to code the random variable a with the distribution $\{q_m\}$

$$q_m = \sum_{mr < i < (m+1)r} p_i = \sum P(m\varepsilon\tau_l < x < (m+1)\varepsilon\tau_l)$$

For the entropy of q we have:

$$H(q) = -\sum_m q_m \log q_m = -\sum_i p_i \log p_i - \sum_{mr < i < (m+1)r} p_i(\log q_m - \log p_i) =$$

$$H(p) + \sum_m q_m \sum_{mr < i < (m+1)r} \frac{p_i}{q_m} \log\frac{p_i}{q_m} =$$

$$H(p) - \sum_m q_m H(p|q_m) = H(p) - H(p|q) < H(p)$$

where $H(p \mid q)$ is the conditional entropy of the distribution $\{p_i\}$ relative to the distribution $\{q_m\}$ [1].

8. Application to AIMS Traces

We calculated the entropy of events, entropy of time stamps and communication entropy of AIMS traces of 4 fluid dynamic programs. The result are shown in Tables 2-3. The basic data about the programs are listed in Table 1.

Small entropy relative to the trace size indicates that a significant number of patterns exists in program traces. For example, a small value of communication entropy in the column 3 of Table 3 indicates that processors send messages in a very regular pattern. The Table 4 compares a theoretial lower bound of a trace code length with length of the code generated by `compress` and `gzip` utilities.

TABLE 1. Basic data on the programs

Program name	t1	t2	t3	t4
Number of Processors	10	27	11	64
Communication Library	MPI	PVM	PVM	MPI
Total Execution Time (s)	313.3	1987.3	1776.0	11.4

TABLE 2. Time stamps entropy as function of time precision

	t1	t2	t3	t4
10%	457	2588	1074	1273
5%	653	3985	1323	3831
2%	958	6510	1645	9062
1%	1234	8748	1899	15578
0.5%	1517	11335	2126	20730

TABLE 3. Minimal code lengths

	t1	t2	t3	t4
Send code length	301	290	46	23052
Receive code length	196	290	99	23052
Event code length	831	8748	3914	58295
Markov chain state number	4519	4234	4261	3485

TABLE 4. Comparison of Sizes of Different Trace Encodings

	t1	t2	t3	t4
Entropy minimal code length (0.5% time approximation)	30897	422879	61453	2867455
Actual trace size	278544	1999818	378893	15269659
Number of records	7013	96104	13817	685581
UNIX compressed	82341	639297	124001	3505437
gziped	60804	482948	93358	3037194

9. Conclusions

In this paper we have studied the information content of traces of message passing programs. The information content is measured as the sum of the entropy of the trace events, the entropy of program communications and the entropy of time stamps.

The information content of program traces is significantly smaller than the trace size. There are several reasons for this: (1) event entropy is related to the entropy of the Markov chain of the program, (2) the messages of common programs have well defined patterns, and (3) the time stamp resolution can be decreased significantly without affecting the event causality. Indeed, we found by direct calculation of the entropy of traces of 4 programs that the entropy is smaller by a factor of 5 than the trace size. This also indicates that a significant number of patterns exists in thouse program traces.

A practical conclusion of this study is that an entropy efficient encoding of program traces will result in significant reduction of trace sizes.

10. References

[1] R. B. Ash. *Information Theory,* Dover, N.-Y., 1990.

[2] A.V. Aho, R. Sethi, J.D. Ullman. *Compilers. Principles, Techniques, and Tools.* Addison-Wesley, 1988.

[3] S. L. Graham, P.B. Kessler, M.K. McKusick. *An Execution Profiler for Modular Programs.* Software-Practice and Experience, Vol 13, 671-685 (1983).

[4] Jerry Yan, Melisa Schmidt, Cathy Schulbach. *The Automated Instrumentation and Monitoring System (AIMS).* Report NAS-97-001, January 1997.

[5] MPI: *A Message-Passing Interface Standard.* June 12. 1995.

[6] Jerry Yan,Henry Jin, Melissa Schmidt. *Performance Data Gathering and Representation from Fixed Size Statistical Data.* NAS Report NAS-98-003.

[7] D.M. Cvetkovic, M.Doob, H. Sachs. *Spectra of Graphs.* V.E.B. Deutscher Verlag Wissenschften, 1979.

[8] J.H. Van Lint, R.M. Wilson. *A Course in Combinatorics.* Cambridge Univ. Press., 1996.

[9] D.E. Knuth. *Dynamic Huffman Coding.* J. of Algorithms, 6:163-180, 1985.

[10] D. Salomon. *Data Compression,* Springer-Verlag, N-Y, 1998.

[11] B.P. Miller, M.D. Callaghan, J.M. Cargille, J.K. Hollingsworth, R.B. Irvin, K.L. Karavanic, K. Kunchithapadam, and T. Newhall. *The Paradyn Parallel Performance Measurement Tools.* IEEE Computer 28 (11), November 1995, pp. 37-46.

[12] G.H. Golub, C.F. Van Loan. *Matrix Computations.* The Johns Hopkins Univ. Press, 1996.

[13] M. Frumkin, R. Hood, L. Lopez. *Trace-Driven Debugging of Message Passing Programs.* IPPS/SPDP 1998, Proceedings, March 30-April 3, 1998, Orlando, pp. 753-762.